The SCC Programmer's Guide

Revision 1.0

Please read the SCC Documentation Disclaimer on the next page.

IMPORTANT - READ BEFORE COPYING, DOWNLOADING OR USING

Do not use or download this documentation and any associated materials (collectively, "Documentation") until you have carefully read the following terms and conditions. By downloading or using the Documentation, you agree to the terms below. If you do not agree, do not download or use the Documentation.

USER SUBMISSIONS: You agree that any material, information or other communication, including all data, images, sounds, text, and other things embodied therein, you transmit or post to an Intel website or provide to Intel under this agreement will be considered non-confidential ("Communications"). Intel will have no confidentiality obligations with respect to the Communications. You agree that Intel and its designees will be free to copy, modify, create derivative works, publicly display, disclose, distribute, license and sublicense through multiple tiers of distribution and licensees, incorporate and otherwise use the Communications, including derivative works thereto, for any and all commercial or non-commercial purposes.

THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT ANY EXPRESS OR IMPLIED WARRANTY OF ANY KIND INCLUDING WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. Intel does not warrant or assume responsibility for the accuracy or completeness of any information, text, graphics, links or other items contained within the Documentation.

IN NO EVENT SHALL INTEL OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, OR LOST INFORMATION) ARISING OUT OF THE USE OF OR IN-ABILITY TO USE THE DOCUMENTATION, EVEN IF INTEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS PROHIBIT EXCLUSION OR LIMITATION OF LIABILITY FOR IMPLIED WARRANTIES OR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER LEGAL RIGHTS THAT VARY FROM JURISDICTION TO JURISDICTION.

*Other names and brands may be claimed as the property of others.

Revision History for Document

0.6	Rewrote the Power Management section to describe new version of the Power Management API.
0.61	Rewrote the section on Building SCC Linux. The sources already exist on MCPC and do not need to be downloaded, and their directory structure does not contain rck.
0.62	Clarified terminology in the first two sections. Updated for sccKit 1.2.0.
0.63	Corrected description of –p pssh switch. Provide link to public SCC SVN repository. Provide link to SCC Bugzilla database.
0.64	Updated for sccKit 1.2.3.
0.65	Updated information about power management
0.70	Changed the start index for voltage levels.
0.72	Cut/paste code errors in Appendix
0.73	Added shared memory calls to table
0.74	Bug in processorID calc in readTILEID.c
0.75	Correct the description of sccReset
0.80	Updated the first 4 sections
0.85	Updated sections 5, 6, and 7
0.86	Bug 256
0.9	Updated section 8, removed section 9, renamed 10 to 9 and 11 to 10
1.0	Updated section 8; added information about the emulator.

Table of Contents

List of Tables

List of Figures

1 Introduction

The Single-chip Cloud Computer (the SCC) is a research chip created by Intel Labs to study many-core CPUs, their architectures, and the techniques used to program them. It has 24 dual-core tiles arranged in a 6x4 mesh. Each core is a P54C core and hence supports Intel architecture. For an overview of the SCC platform, refer to the *SCC Platform Overview*.

The SCC hardware is a circuit board that contains the SCC chip, memory, and a system interface. The SCC usage model will evolve over time, but currently two platforms are supported.

The first platform is the Linux platform. It runs a version of opensource Linux on each core. You load your application on one or more cores and the program runs with operating system support, much as you'd expect on a node in a cluster. The other platform is the baremetal platform. Here the cores do not have an operating system. Your application runs directly on the cores without any operating system support. At this stage in SCC's lifetime, the Linux platform described in Section 2 is the most mature.

Section 3 SCC Architecture and Performance Considerations defines some terminology used when configuring and programming the SCC. It also discusses some performance considerations. Key SCC features are a large address space and a large number of IA (Intel Architecture) cores that support a message-passing programming model. A unique feature of the SCC is its ability to adjust the voltage and frequency of the tiles, both at startup and dynamically during operation. Refer to Section 5 Power Management.

With the current usage model, you connect a PC called the Management Console PC (MCPC) to the system interface on the SCC platform. The MCPC runs a version of Ubuntu, greater than 10.0. A typical MCPC has Ubuntu 10.04.3 LTS. Refer to Section 4 The Management Console .

You then use Intel-provided software that runs on the MCPC to configure the SCC platform, compile your application, and then load your application on the SCC cores. Your application's I/O is through the MCPC. Section 4 The Management Console also describes the software that runs on the Management Console.

However, it is important to note that this is only an *initial* usage model. Intel Labs has designed the SCC with flexibility and potential in mind. How you actually use the SCC platform is determined by your own imagination and experience.

When you compile programs that run on the MCPC, you typically use gcc/g++. When you compile for the SCC platform, you should use an older version of icc/icpc or gcc/g++. The Intel compiler you need is an older version because the SCC cores are based on the P54C architecture, the Pentium® architecture before the introduction of the streaming SIMD extensions (SSE) and out-of-order execution.

This document assumes that you are an experienced parallel programmer. Most likely you already have a parallel application that you are interested in porting to the SCC. Some of the issues you may face when porting to the SCC platform are the lack of caching coherence among the cores and the idiosyncrasies of the older P54C architecture.

This document also shows how to run some very simple "from-scratch" programs on the SCC platform. These programs range from a simple "hello-world" to short programs that

read and write SCC configuration registers. Section 6 Some Simple SCC Programs describes how to write, compile, and run such simple programs.

Intel also provides RCCE. RCCE (pronounced "rocky") is a many-core communication environment for SCC application programmers. With RCCE, you can write message-passing application programs for either the Linux or baremetal platforms. The RCCE package also contains a number of sample applications, ranging from simple (such as two cores just exchanging messages) to complex (such as a subset of the NAS parallel benchmarks). Section 8 Running RCCE Applications describes how to run some of these applications.

Section 7 Building RCCE describes how to build RCCE. You can build RCCE as an emulator or as a library intended for SCC hardware. When you use RCCE as an emulator, you can create RCCE applications that run on any standard Linux or Windows computer not connected to the SCC platform. The RCCE emulator is built on top of OpenMP.

The RCCE library is useful and highly performing in and of itself, but Intel also provides the complete source code for RCCE. You can look at how RCCE does what you want to do and write your own versions.

Please make yourself a Bugzilla account on http://marcbug.scc-dc.com/bugzilla3/. If you experience a usage issue with the SCC, you may find a solution in this Bugzilla database. In addition, please enter bugs for new issues you may experience. The SCC support team monitors this Bugzilla database very closely.

2 The Linux Platform

The MCPC contains an Intel-provided Linux image that runs on the SCC cores. You can build and use your own Linux image modeled after the one that Intel provides.

SCC Linux has been designed to run on the SCC cores. It will evolve as the SCC platform develops. For example, currently, I/O calls in a core program are redirected to a memory-mapped interface, and the output then appears at the Management Console.

You run the sccGui to configure the SCC platform. Configuring means training the system interface and the DDR3 memory on the SCC platform, setting values in SCC configuration registers, and loading Linux on one or more cores.

You can also perform these configuration actions from a command line on the MCPC.

3 SCC Architecture and Performance Considerations

Each of the 24 tiles has two cores. Each core has L1 and L2 caches. Each core has a 16KB LI instruction cache and a 16KB L1 data cache. The L1 caches are on the core. Each core also has a 256KB L2 cache. The L2 caches are on the tile.

Each tile also has a message passing buffer (MPB). This message passing buffer is 16KB of SRAM, local to the tile. This memory is shared among all the cores on the chip. Each tile has its own local MPB. Conventionally, it assigns 8KB to one of its cores and 8KB to the other. Although assigned to a particular core, the MPB is accessible to all cores. This

convention is configurable. The total MPB for all the tiles is 384KB.

When a message-passing program sends a message from one core to another, internally it is moving data from the L1 cache of the sending core to the MPB and then to the L1 cache of the receiving core. The MPB allows L1 cache lines to move between cores without having to use the off-chip memory.

The off-chip DRAM is accessed through four on-die memory controllers. This off-chip DRAM is divided into memory private to each core and memory shared by all cores. The maximum off-chip DRAM is 64GB. A core has a 32-bit address space and hence can address 4GB. A core accesses a 32-bit *core address* which must be translated into a *system address*.

Each core also has a lookup table (LUT). This LUT translates the core address into a system address. A core also accesses its configuration registers via memory-mapped I/O, and the LUT translates a core address into the addresses needed for memory-mapped I/O.

System memory is divided into memory private to a core and memory shared by all the cores. Where this division occurs is determined by the core's LUT and the core's own pagetables. When you load SCC Linux on the cores, the LUTs are filled with default values. Refer to the *SCC EAS* for a listing of these default values. You can modify the values in the LUT dynamically after loading SCC Linux.

Message-passing data are typed as message-passing buffer type (MPBT). Data typed as MPBT bypass the L2 cache. If you write to data that are already resident in the cache, the cache line may (if L1 is configured as write-through) or may not (if L1 is configured as write-back) be moved to memory.

It is important to note that there is no cache coherence protocol among the cores. This means that when a core reads MPBT data, it gets the values in the L1 cache even when the data are stale. Cores may not get the latest MPBT data unless they invalidate the MPBDT data in their own LI cache.

To solve this problem, the SCC has a new instruction called CL1INVMB. This instruction invalidates all lines in the L1 cache that contain message buffer data, that is, data typed as MPBT.

A mesh interface unit (MIU) on each tile catches a cache miss and then using the LUT to decode the core address into a system address. If the data are typed MPBT, this is an L1 cache miss.

The LUT translates a 32-bit core address into a 46-bit system address. The most significant bit (bit 45) of the system address is the bypass bit. Ignore the bypass bit; do not use it. There is a hardware bug with the bypass bit that will not be fixed.

The next eight bits (bits 44 through 37) determine the tile, which might be the same tile whose core is requesting access. The next three bits (bits 36 through 34) determine whether the access is to a non-local MPB, a configuration register, a memory controller, or the system interface. When the access is to off-chip DRAM, the lower 34 bits (bits 33 through 0) go to the memory controller on the specified tile. Each memory controller uses those 34 bits to address up to 16GB.

Messages employ XY routing. Messages go from the sending core to the receiving core first along the X direction and then along the Y direction. When this rule is followed, note that

when a message goes back and forth between cores, the “back” path is different from the “forth” path.

3.1 Latencies

Table 1 lists some approximate latencies experienced when a core reads a 32-byte cache line. The table shows both core cycles and mesh cycles. A core and the mesh may run at the same frequency, but because of the SCC’s power management capability, these frequencies may also be different. It is even possible that individual cores may run at different frequencies. In the table, core cycles refers to the cycles of the core making the request.

Latency Table	Approximate latency to read a cache line (output from the core to input back to core)
L2 access	18 core cycles
Local MPB access with bypass	15 core cycles
Local MPB access no bypass	45 core cycles + 8 mesh cycles
Remote MPB access	45 core cycles + 4*n*2 mesh cycles
DDR3 access	40 core cycles + 4*n*2 mesh cycles + 30 on-die memory controller (400MHz)+ 16 cycles(400MHz off-die DDR3 latency)
	n=number of hops to the MPB or the memory controller (0<n<10)

Table 1: Core Latency Table

3.2 processorID, tileID, and coreID

There are three IDs associated with a core: the processorID, the tileID, and the coreID. In Table 2, the number in the lower left of each tile is the tileID. The (x,y) coordinates are in the lower right.

You can get this information by reading the TileID register. The lower 11 bits of this register are valid. Bits 10:07 contain the Y value; bits 05:03 contain the X value. Bits 02:00 contain the subID of the requesting agent. If the requesting agent is a core, its subID is the coreID.

If instead of running a program on the core to access the TileID register, you are using the sccGui, the coreID is always 101b. When you are using the sccGui, the requesting agent is the system interface whose subID is 101b.

Note that a core’s tileID is not the same as the value of a core’s TileID register. The tileID is an 8-bit value; the TileID register has 11 valid bits.

Because a core's tileID is an 8-bit hex value, it is not continuous. Its value is just the upper eight bits of the TileID register. If you have a core's (x,y) coordinates, get the tileID as 0x*yx*. Numerically, this is tileID = 16*y + x.

Calculate the processorID as ((x + (6 * y)) * 2) + coreID. The processorID goes from 0 through 47.

37 36 0x30 (0,3)	39 38 0x31 (1,3)	41 40 0x32 (2,3)	43 42 0x33 (3,3)	45 44 0x34 (4,3)	47 46 0x35 (5,3)
25 24 0x20 (0,2)	27 26 0x21 (1,2)	29 28 0x22 (2,2)	31 30 0x23 (3,2)	33 32 0x24 (4,2)	35 34 0x25 (5,2)
13 12 0x10 (0,1)	15 14 0x11 (1,1)	17 16 0x12 (2,1)	19 18 0x13 (3,1)	21 20 0x14 (4,1)	23 22 0x15 (5,1)
1 0 0x00 (0,0)	3 2 0x01 (1,0)	5 4 0x02 (2,0)	7 6 0x03 (3,0)	9 8 0x04 (4,0)	11 10 0x05 (5,0)

Table 2 The SCC Mesh Showing tileIDs, processorIDs, and (x,y) Coordinates. The tileID is best shown as an 8-bit hex value, with the core's y-coordinate in the upper four bits and the core's x-coordinate n the lower four bits; this convention means that the tileID is not numerically continuous.

4 The Management Console

The Management Console is a PC that communicates with the SCC platform over a PCIe bus. The PCIe bus connects to a System FPGA interface on the SCC board which connects to a System Interface on the SCC itself. Users typically VNC into the Management Console from their own workstation.

The management console (MCPC) runs Ubuntu Linux. Currently, most MCPCs are running Ubuntu 10.04.3 LTS.

You should ensure that your MCPC has **`pssh`** and Python. You can use **`pssh`** to load programs on the cores or execute commands on the cores. The script **`rccerun`** that loads RCCE applications on the cores uses **`pssh`** internally.

4.1 How to Get Copies of the Latest MCPC Tools

The public SVN repository is located at http://marcbug.scc-dc.com/svn/repository/. This repository has anonymous read.

Click on **`tarballs`**. A list of available MCPC tools appear as compressed tar files as shown in Figure 1. The file list you see may be different as files are added or removed. Currently, the latest version of sccKit is 1.4.1.3. Click on a file to download it.

Figure 1: Available MCPC Tools

Some source code is also available. Instead of tarballs, click on trunk or tags. The tags directory contains snapshots of the trunk labeled with a specific release number. The trunk contains the very latest code, which in most cases is equal to the latest tag.

4.2 Installing sccKit

The very latest instructions on how to install sccKit are on the Marc coummunity site in the subcommunity called *Your MCPC* http://communities.intel.com/community/marc/yourmcpc in the *How To* section. Click on the Documents tab or click on the name of the document in the *How To* pane. This section gives an overview of the installation. Please look at the website for complete instructions.You need to have root access to be able to install sccKit.

The sccKit is distributed as two tar files: **`sccKit_base.tar.bz2`** and **`sccKit_1.4.1.3.tar.bz2`**. As of September 5, 2011, sccKit 1.4.1.3 is the latest version.

- **`sccKit_base.tar.bz2`** contains the basic sccKit components and will change infrequently. It also contains the Qt libraries that sccKit needs. If you install the Qt SDK, you don't need the libraries from this tar file because you already have them.
- **`sccKit_1.4.1.3.tar.bz2`** contains the actual sccKit release.

The MCPC has a driver called **`crbif`** that implements communication with the SCC. The **`crb`** stands for Copperridge Board. The SCC chip curently resides on the RockyLake board; the Copperridge board is an older version. The **`crbif`**driver has two components:

- The PCIe interface to the sccKit software components (for example, the sccGui and sccKit commands)
- The Ethernet device that communicates with the SCC core (Ethernet over PCIe or Ethernet over eMAC).

Prior to sccKit 1.4.0, only the PCIe connection and Ethernet over PCIe were supported. Beginning with 1.4.0, sccKit supported the eMAC connection. Figure 2 shows back of the SCC unit with the PCIe and Ethernet connections.

There are four eMAC connections, labeled left to right as ABCD. You cannot enable all four. You can enable AB or CD. Not all SCC units have four functional eMAC connections. Most SCC installations enable only one eMAC connection. Some SCC units only have one functional eMAC connection. The Ethernet cable in Figure 2 is connected to eMAC D.

When you install sccKit 1.4.x, you need to add a Gigibit switch to the installation. Figure 3 illustrates how the SCC and the MCPC are connected. The IP addresses in the figure are just examples. The figure shows two Ethernet cables coming from the SCC chassis. The MCPC also has two Ethernet cables and two NICs. The **`eth0`** cable connects to the Internet, most likely through your own router and firewall. The **`eth1`** cable connects to the BMC. Typically, users configure **`eth1`** to have a virtual ethernet connection **`eth1:1`** as well. This allows the BMC and the MCPC to be on the same subnet so that you can telnet into the BMC from the MCPC.

.

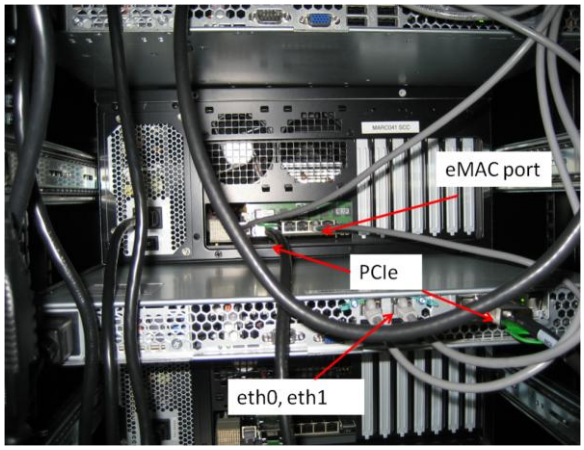

Figure 2: The Back of the SCC Unit, Showing the PCIe and eMAC Connections

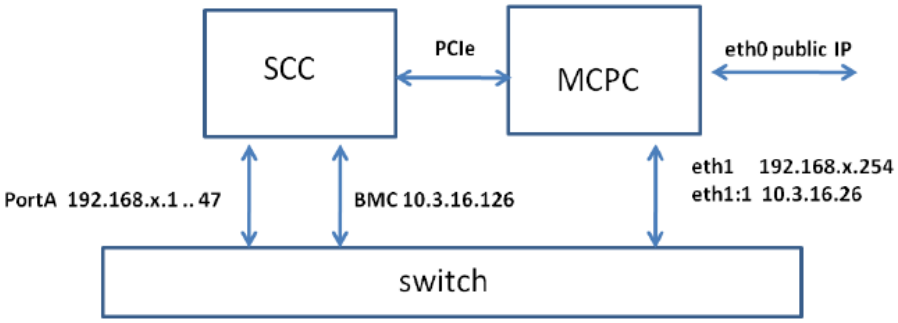

Figure 3: An SCC/RockyLake System Showing the Connection for the Ethernet Cables and the PCIe cable

sccKit 1.4.0, however, experienced a bug related to eMAC, http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=264 . SCC cores would lose connectivity, then regain it, then lose it again. The system remained functional if eMAC was disabled. This bug was fixed in sccKit 1.4.1, but this release turned out to be unstable.

You should run at least sccKit 1.4.1.2. This release is 1.4.1 with patches 1 and 2, sequentially applied; patch 2 does not contain patch 1, and pathc1 must be applied before patch 2. Patch 3 corrects a production test program that does not affect user operation.

- sccKit 1.4.1.3 does have some outstanding bugs. Beta fixes exist and will either be incorporated into a patch 4 or a new release.
- sccWrite See http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=304
- sccGui does not update the clock configuration register (although the reigster's is actually changed). See http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=271

Even when the CPU frequency is changed, SCC Linux still reports 800 MHz. Consequently SCC Linux system calls return the wrong value. This is also part of http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=271

Check that the file **`systemSettings.ini`** in sccKit's top-level directory contains the correct IP address of the SCC BMC. This address is printed on the top of the BMC. The port is always 5010. You can telnet to the BMC as follows. Use your own BMC IP address.

```
telnet 172.28.248.241 5010
```

To access the sccKit binaries, put **`/opt/sccKit/current/bin`** in your path. You also need to set the environment variable **`LD_LIBRARY_PATH`** to /**`opt/sccKit/lib`** so that you can access the shared Qt libraries. The script **`setup`** in sccKit's top-level directory sets your path and **`LD_LIBRARY_PATH`** for either the bash or tcsh shells.

sccKit also contains an integrated production test. You need to be **`root`** to run this command. You need to have patch 3 installed. If you are using the SCC through the Intel Labs data center, do not run the test. Data center managers have that responsibility. You can choose to run the test on a local MCPC, when you install a new SCC board and after you run **`install.sh`** successfully.

```
sccProductionTest
```

4.3 Using sccKit

The sccKit has both a GUI and a command line. The capabilities of the sccGui overlap significantly but not completely with those available from the command-line.

For example, you can re-initialize the SCC platform by issuing the command, **`sccBMC -i`**. You can also bring up sccGui, click on the BMC tab, and choose (Re-)initialize platform.

To invoke the SCC GUI, issue the command, **`sccGui&`**. Figure 4 shows the initial SCC GUI screen.

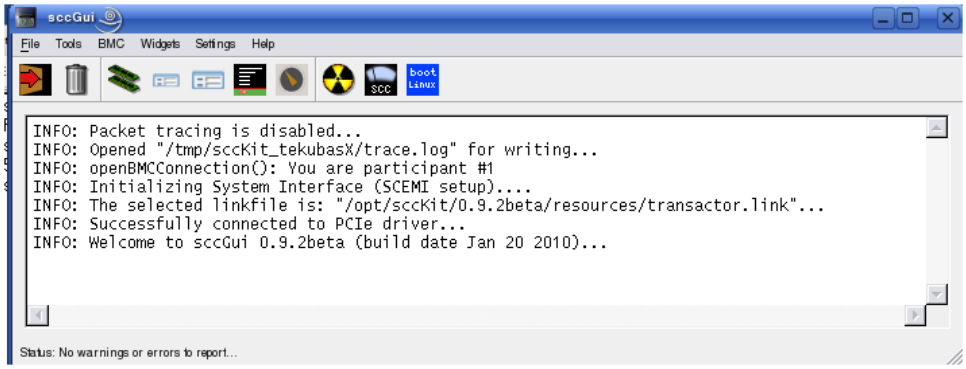

Figure 4: The SCC GUI

Table 3 lists the SCC functions available from the command line. Each command has online help. Issue the command with the **`-h`** option to get help information.

SccKit Command	Description
sccBMC	Initialize the SCC platform. Send commands to the BMC.
sccBoot	Boot Linux on one or more cores.
sccCmdline	Patch either the merged ore pre-merged linux image with command-line parameters.
sccDisplay	Start a virtual display with one tab per available core. Needs desktop (even for –h).
sccDump	Specify a tileID as in *yx* format and read memory or memory-mapped registers from that tile.
sccKonsole	Start up a Konsole on one or more cores. Needs a desktop.
sccMerge	Creates a merged obj file, for each memory controller.
sccReset	Reset one or more cores. You can reset/release, reset, or release one or more cores.
sccPerf	Display the SCC performance meter. Needs a desktop.
sccPowercyle	Performs a hard SCC power cycle. Load a new FPGA bitstream
sccWrite	Writes memory content (or memory-mapped registers) to the SCC.

Table 3 sccKit Commands

You can bring up SCC Linux on the cores with either **sccGui** or the command line. To load SCC Linux with the SCC GUI, click on the blue Boot Linux button. Another window comes up that allows you to choose the cores on which you want to load SCC Linux. The default location of the SCC Linux image is **/opt/sccKit/current/resources**. You can change this default location by selecting Settings➔Linux boot settings➔Choose Linux image, as shown in Figure 5.

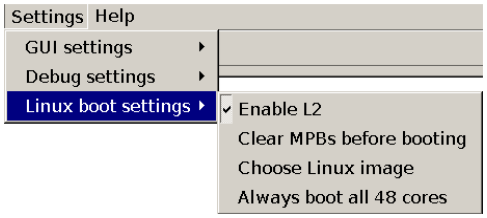

Figure 5: Changing the SCC Location with SCC GUI

When the SCC Linux boot is successful, the blue Boot Linux button changes to a green Linux okay button.

You can also boot SCC Linux from the command line with **sccBoot**. Without arguments, you just get the help message. Boot SCC Linux with the **-l** option. Use the **-g** option to specify a custom obj directory. For example, the command below loads **mylinux.obj** on cores 0 through 7.

```
sccBoot -l 0..7  /home/username/mylinux.obj
```

With the **-s** option, you can check that the cores have successfully booted.

```
username@mrllab1002:~$ sccBoot -s
INFO: Welcome to sccBoot 1.1.0 (build date Apr 29 2010)...
Status: The following cores can be reached with ping (booted): 8 cores
(PIDs = 0x00, 0x01, 0x02, 0x03, 0x0c, 0x0d, 0x0e and 0x0f)...
username@mrllab1002:~$
```

You can also ping the SCC cores. The cores are called `rck`*pid* where pid is the core's pid and goes from 0 to 47. The pid here is the same as the processorID shown in Table 2.

```
username@mrllab1002:~$ ping rck01
PING rck01.in.rck.net (192.168.0.2) 56(84) bytes of data.
64 bytes from rck01.in.rck.net (192.168.0.2): icmp_seq=1 ttl=64
time=0.240 ms
```

With sccGui, you can look at the performance meter. Bring up the performance meter by selecting Widgets➔SCC Performance Meter as shown in Figure 6. You can also start up the SCC performance meter from the command line with **sccPerf**. Note, however, that sccPerf must connect to an X server. You cannot run it from an ssh window; either work directly on the MCPC or VNC into the MCPC (the more common method).

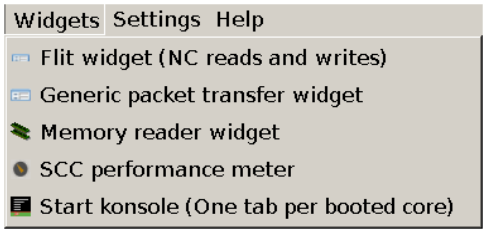

Figure 6: Selecting the SCC Performance Meter

You can tell that SCC Linux is running on a core if the core has a green arrow. Figure 7 shows the SCC performance meter with eight green arrows because SCC Linux is running on eight cores in the lower left 2x2 tile array.

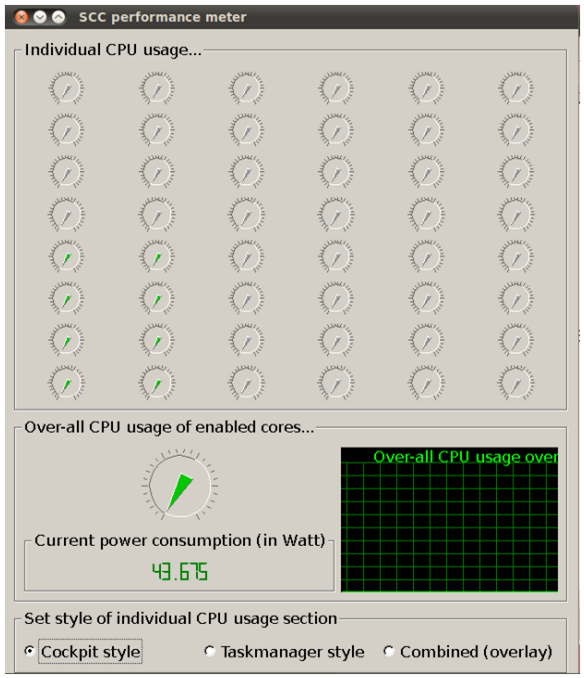

Figure 7: The SCC Performance Meter

When you click on the green Linux okay button, you can restart SCC Linux on the cores currently running SCC Linux. The "Boot Linux on selected cores …" window comes up with the cores currently running SCC Linux selected.

```
username@marc101:~$ sccBoot -s
INFO: Welcome to sccBoot 1.4.1 (build date Jul  4 2011 - 16:14:13)...
Status: The following cores can be reached with ping (booted): 8 cores
(PIDs = 0x00, 0x01, 0x02, 0x03, 0x0c, 0x0d, 0x0e and 0x0f)...
```

You can choose at this point to reboot the cores that are running SCC Linux or to add additional cores.

If you deselect the cores in the lower left 2x2 array and select those in its adjacent 2x2 array, you boot SCC Linux on eight more cores. SCC Linux continues to run on the original eight

cores, and so now you have SCC Linux running on 16 cores.

```
username@marc101:~$ sccBoot -s
INFO: Welcome to sccBoot 1.4.1 (build date Jul  4 2011 - 16:14:13)...
Status: The following cores can be reached with ping (booted): 16
cores (PIDs = 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x0c,
0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12 and 0x13)...
```

Note that booting on additional cores does not affect the cores that already have SCC Linux running. To stop SCC Linux from running on a core, reset that core. To do that, select Tools➔System reset as shown in Figure 8 . By default, you reset all the cores. Select cores to reset with Tools➔Change selected reset(s).

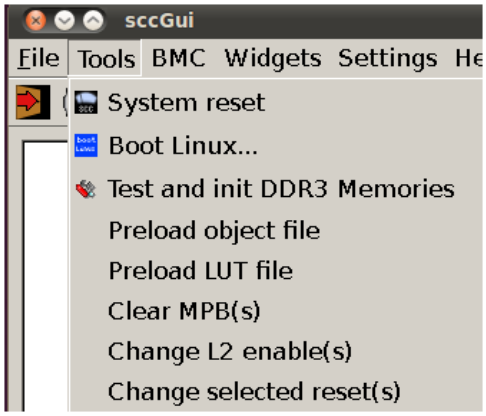

Figure 8 Selecting System Reset

With the command line, you can choose to reset individual cores. You can see what cores are in reset with the **-s** option. For example, the command below shows 32 cores in reset. This is because we have SCC Linux running on the two 2x2 arrays in the lower row; SCC Linux is running on 16 cores, leaving 32 in reset.

```
username@marc101:~$ sccReset -s
INFO: Welcome to sccBoot 1.4.1 (build date Jul  4 2011 - 16:14:13)...
Status: The following resets are active (pulled): 32 cores (PIDs =
0x08, 0x09, 0x0a, 0x0b, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a,
0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25,
0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e and 0x2f)...
```

For example, you can choose Tools➔Change selected reset(s). You will see all the cores selected except for those that have Linux running. You can then select cores 0..3 and click Okay. Note that now four more cores are in reset.

```
username@marc101:~$ sccReset -s
INFO: Welcome to sccBoot 1.4.1 (build date Jul  4 2011 - 16:14:13)...
Status: The following resets are active (pulled): 36 cores (PIDs =
0x00, 0x01, 0x02, 0x03, 0x08, 0x09, 0x0a, 0x0b, 0x14, 0x15, 0x16,
0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21,
0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c,
0x2d, 0x2e and 0x2f)...
```

You can also use the command line. The **-p** resets the core and holds the reset. The pid numbers can be in decimal or hex, but when in hex, they should be preceded with 0x.

```
username@marc101:~$ sccReset -p 0xc..0xf
INFO: Welcome to sccBoot 1.4.1 (build date Jul  4 2011 - 16:14:13)...
INFO: Resets have been pulled: 4 cores (PIDs = 0x0c, 0x0d, 0x0e and
0x0f)...
username@marc101:~$ sccReset -s
INFO: Welcome to sccReset 1.3.0 (build date Aug 25 2010 - 15:55:54)...
Status: The following resets are active (pulled): 40 cores (PIDs =
0x00, 0x01, 0x02, 0x03, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
0x0f, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d,
0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28,
0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e and 0x2f)...
```

We now have put 8 cores into reset and have Linux running on 8 cores.

The **sccReset** command also gives you the option of releasing the reset with the **-r** option. Releasing the reset will not reboot Linux.

The example below releases the reset on cores 0 though 3. Note that we have 36 cores in reset. Linux is running on 8 cores. We released reset on 4 cores, but have not rebooted Linux on those cores..

```
username@marc101:~$ sccReset -r 0..3
INFO: Welcome to sccBoot 1.4.1 (build date Jul  4 2011 - 16:14:13)...
INFO: Resets have been released: 4 cores (PIDs = 0x00, 0x01, 0x02 and
0x03)...
username@marc101:~$ sccReset -s
INFO: Welcome to sccBoot 1.4.1 (build date Jul  4 2011 - 16:14:13)...
Status: The following resets are active (pulled): 36 cores (PIDs =
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x14, 0x15, 0x16,
0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21,
0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c,
0x2d, 0x2e and 0x2f)...
```

What does it mean to release reset on a core? It means that the core starts running. Letting the core run is different than starting Linux. The **sccReset -r** started the boot process on a "used" version of Linux with unknown results. However, the command turns out to be useful for baremetal applications and for applications using mixed operating systems. You would use it in combination with the sccGui entry Tools➔Preload object file.

If you are thinking of investigating mixed operating systems, please look at http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=254 .

With sccKonsole, you can open up an ssh connection to one or more cores. To start a console on cores whose pids are 0,1,2,3, issue

```
sccKonsole 0..3
```

Each tabbed konsole is a separate connection. If you want input from one konsole to be recognized by another, select Edit➔Copy Input To➔All Tabs in Current Window. Figure 9 shows the four tabbed windows created with sccKonsole 0..3. The konsole **rck01** was configured to send its input to all konsoles; that's why the red !. The konsole **rck01** is selected and shows an **ls -a** command that was issued in **rck01**.

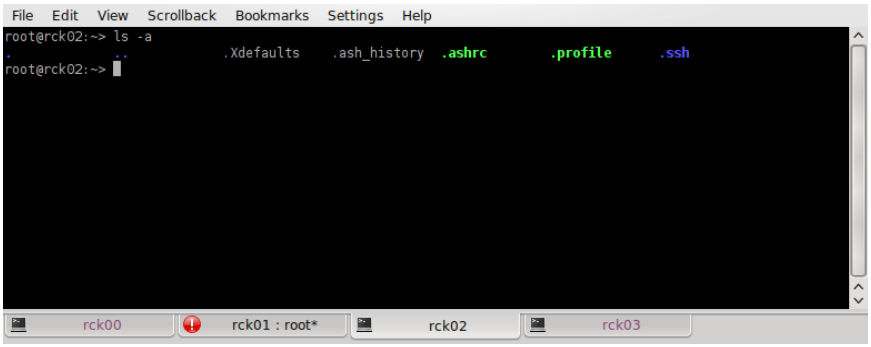

Figure 9: Result of sccKonsole 0..3

You can also create tabbed konsoles with sccGui. Bring up tabbed konsoles by selecting Widgets➔Start konsole (one tab per booted core) as shown in Figure 6. With the SCC GUI, you always create one tab per booted core. Use the command line if you want to select what cores get a konsole. As with **sccPerf**, **sccKonsole** requires connection to an X server.

The command **sccDump** is useful when debugging. With **sccDump** you can read memory (off-chip DRAM), the message-passing buffer, the configuration registers, and the lookup tables. It corresponds to the memory reader widget in sccGui. Select the memory reader widget with Widgets➔Memory reader widget as shown in Figure 6.

For example, to display the configuration registers and lookup tables for the tile in the upper left corner and save the output to a file called **sccDump_c_0x31.txt**, enter

```
sccDump –c 0x30
```

sccDump identifies the tile with its tileID. Recall that the tileID is an 8-bit number with the y coordinate in the upper four bits and the x coordinate in the lower four bits; the tileID is not continuous over the mesh. Instead of 0x30, you could have specified the decimal number 38.

The output of sccDump looks as follows:

```
INFO: Packet tracing is disabled...
INFO: Initializing System Interface (SCEMI setup)....
INFO: Successfully connected to PCIe driver...
INFO: Welcome to sccDump 1.4.1 (build date Jun 28 2011 - 16:02:28)...
============================================================
Dumping CRB registers of Tile 0x30
============================================================
GLCFG0   = 0x00348df8
GLCFG1   = 0x00348df8
L2CFG0   = 0x000006cf
L2CFG1   = 0x000006cf
SENSOR   = 0x00000000
GCBCFG   = 0x00a8e2f0
MYTILEID = 0x00000185
LOCK0    = 0x00000001
LOCK1    = 0x00000001
------------------------------------------------------------
Restoring locks: LOCK0 and LOCK1
============================================================
Dumping LUTs of Tile 0x30
Format: Bypass(bin)_Route(hex)_subDestId(dec)_AddrDomain(hex)
```

```
============================================================
LUT0, Entry 0x00 (CRB addr = 0x0800): 0_0x20_6(PERIW)_0x078
LUT0, Entry 0x01 (CRB addr = 0x0808): 0_0x20_6(PERIW)_0x079
LUT0, Entry 0x02 (CRB addr = 0x0810): 0_0x20_6(PERIW)_0x07a
LUT0, Entry 0x03 (CRB addr = 0x0818): 0_0x20_6(PERIW)_0x07b
LUT0, Entry 0x04 (CRB addr = 0x0820): 0_0x20_6(PERIW)_0x07c
LUT0, Entry 0x05 (CRB addr = 0x0828): 0_0x20_6(PERIW)_0x07d
LUT0, Entry 0x06 (CRB addr = 0x0830): 0_0x20_6(PERIW)_0x07e
LUT0, Entry 0x07 (CRB addr = 0x0838): 0_0x20_6(PERIW)_0x07f
LUT0, Entry 0x08 (CRB addr = 0x0840): 0_0x20_6(PERIW)_0x080
LUT0, Entry 0x09 (CRB addr = 0x0848): 0_0x20_6(PERIW)_0x081
LUT0, Entry 0x0a (CRB addr = 0x0850): 0_0x20_6(PERIW)_0x082
LUT0, Entry 0x0b (CRB addr = 0x0858): 0_0x20_6(PERIW)_0x083
LUT0, Entry 0x0c (CRB addr = 0x0860): 0_0x20_6(PERIW)_0x084
LUT0, Entry 0x0d (CRB addr = 0x0868): 0_0x20_6(PERIW)_0x085
LUT0, Entry 0x0e (CRB addr = 0x0870): 0_0x20_6(PERIW)_0x086
LUT0, Entry 0x0f (CRB addr = 0x0878): 0_0x20_6(PERIW)_0x087
LUT0, Entry 0x10 (CRB addr = 0x0880): 0_0x20_6(PERIW)_0x088
LUT0, Entry 0x11 (CRB addr = 0x0888): 0_0x20_6(PERIW)_0x089
LUT0, Entry 0x12 (CRB addr = 0x0890): 0_0x20_6(PERIW)_0x08a
LUT0, Entry 0x13 (CRB addr = 0x0898): 0_0x20_6(PERIW)_0x08b
LUT0, Entry 0x14 (CRB addr = 0x08a0): 0_0x64_1(CORE1)_0x141
                                    :
                                    :
LUT0, Entry 0xfe (CRB addr = 0x0ff0): 1_0x88_0(CORE0)_0x0a4
LUT0, Entry 0xff (CRB addr = 0x0ff8): 0_0x20_6(PERIW)_0x0fa
LUT1, Entry 0x00 (CRB addr = 0x1000): 0_0x20_6(PERIW)_0x08c
LUT1, Entry 0x01 (CRB addr = 0x1008): 0_0x20_6(PERIW)_0x08d
```

These are the LUT values for tile 0x30. The tile numbers are in 0xYX format, so this is the tile in the upper left corner. The LUT values for memory are shown with four fields: for example, **`0_0x20_6(PERIW)_0x078`**. The first field is always zero (the bypass bit); the second field identifies the location of the memory controller used by the core (the memory controllers are at 0x00, 0x20, 0x05, and 0x50); the third field identifies the direction of the memory controller (6 for West and 4 for East); the fourth field contains the 10 bits prepended to the address. Please refer to the EAS for details,

`sccMerge` reads a configuration **`mt`** file (for example, **`linux.mt`**) and creates merged objects, one for each memory controller. For a discussion about how to use **`sccMerge`** and an example, refer to How to Map Cores to Memory Controllers

4.4 MCPC Tools

In addition to the sccKit, MCPC tools consist of C and Fortran cross compilers, a version of the Math Kernel Library (MKL), and an **`.ssh2`** directory.

You can use any C compiler you want for programs intended to run on the MCPC. Typical C compilers are **`gcc/g++`** and the latest version of Intel's **`icc/icpc`**. The RCCE emulation library builds and runs with either compiler.

The C and Fortran cross compilers are older versions of Intel's compilers that work on the P54C architecture. The P54C architecture is the Pentium architecture before the introduction of streaming SIMD instructions and out-of-order execution. The MKL is also an older version for use on the P54C architecture.

If you are building an application to run on the cores themselves, you must use the Intel-provided cross compilers. Table 4 lists the versions that work with the SCC platform.

Cross Compiler/Library	Version
`gcc/g++`	3.4.5
`icc/icpc`	8.1.038
`ifort`	8.1.034
`mkl`	8.1.1.004

Table 4: Cross Compiler and Library Versions that Run on the SCC Cores

Ensure that you have an acceptable `.ssh2` directory. SCC applications run on the cores as root, and the `.ssh2` directory lets that happen.

The sccKit consists of the sccGUI and some command line utilities. With the sccGui, you can perform such actions as booting Linux on one or more cores, reading memory, and reading/writing configuration registers.

4.5 Installing MCPC Tools

If you are using one of the MCPCs in the Intel Labs data center, the compilers and other tools are already installed.

If you have your own MCPC and SCC unit, please refer to the Marc Community site for instructions How Set up the Licensed Intel Compiler .

5 Power Management

The SCC platform contains a voltage regulator controller (VRC) that allows you to independently change the voltage of an eight-core voltage island. You can also change the frequency of individual cores. You can do this dynamically from within a program running on the cores.

When you initialize the SCC platform, you can choose the initial frequency settings. Do this by selecting the re-initialize button. Refer to Figure 10. The re-initialize button is the black-and-yellow button that looks like a radiation hazard warning.

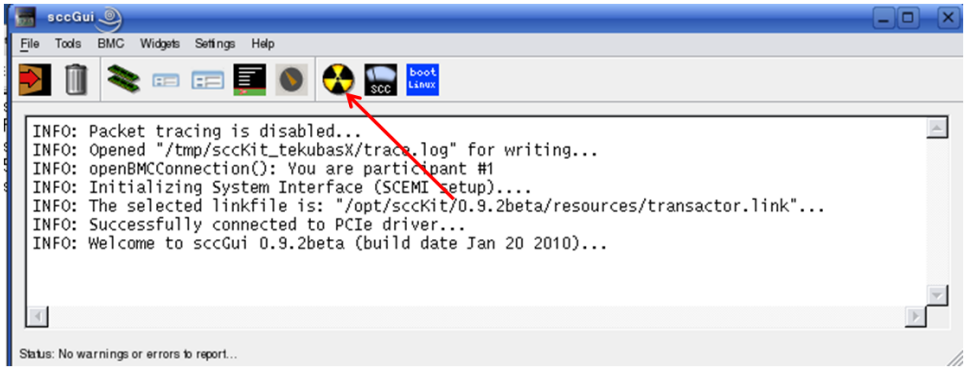

Figure 10: Initialize the SCC Platform

When you click the re-initialize button, you get the screen in Figure 11. With the dropdown box, you can choose from among the following five settings. The numbers refer to clock frequencies measured in MHz. Core is the frequency of the core; router is the frequency of the mesh; and MC is the frequency of the memory. The default is **`Tile533_Mesh800_DDR800`**. It specifies that the cores run at 533MHz and that the mesh and the memory run at 800MHz. The voltage for all settings is nominally 1.1v. Choose **`Tile533_Mesh800_DDR800`** for normal operation.

Tile533_Mesh800_DDR800 (Default)
Tile800_Mesh1600_DDR1066
Tile800_Mesh1600_DDR800
Tile800_Mesh800_DDR1066
Tile800_Mesh800_DDR800

To see the actual voltages for the power domains, you can telnet into the BMC from the MCPC, specifying port 5010. Then, issue the status command.

Console. From a prompt on the Management Console, type the command

```
telnet <BMC IP address> 5010
```

The BMC IP address is assigned to the platform when you receive it. The default value is 192.168.2.127. It's most likely written on a sticker attached to the BMC.If this is a Data Center system, we changed this value. The BMC IP address is shown in **`/opt/sccKit/systemSettings.ini`**. For information about how to change the BMC IP address, refer to http://communities.intel.com/docs/DOC-5592 .

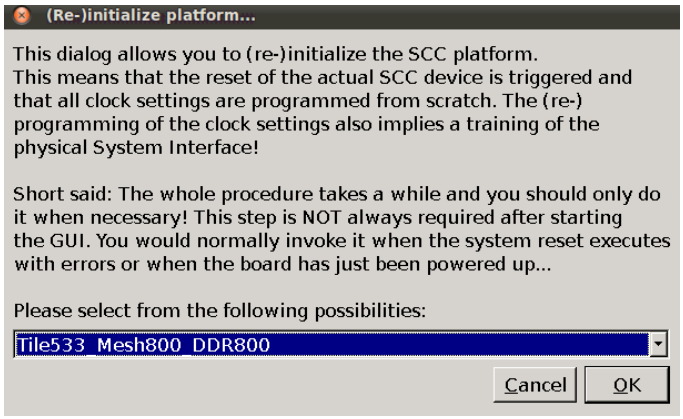

Figure 11: Choosing a Voltage/Frequency Setting

The default is **`Tile533_Mesh800_DDR800`**, and this is the recommended setting. The other possibilities, shown in the dropdown box, are **`Tile533_Mesh1600_DDR1066`**, **`Tile800_Mesh1600_DDR800`**, **`Tile800_Mesh800_DDR800`**, and **`Tile800_Mesh800_DDR800`**.

The goal is, of course, to improve performance while minimizing power and not killing the chip. Running the cores at a frequency that is too high for the existing voltage may cause the system to crash.

Dynamic power is proportional to frequency * voltage squared. Here are some guidelines.

- If your program is computation CPU bound, its performance is proportional to the frequency.
- Only reduce the frequency for programs that are not computation bound; that is, programs that are accessing I/O or memory.
- Try to minimize both frequency and voltage. If you cannot, minimize the frequency.
- Remember to take into account the latency when changing the voltage. This latency is on the order of milliseconds. The latency for changing the frequency is much smaller, about 20 cycles.

SCC cores are divided into seven voltage domains. Six of those are 2x2 arrays of tiles, as shown in Figure 12. The seventh is the entire mesh. Currently, the RCCE API does not provide the ability to access the "entire-mesh" domain. When you change the voltage, you choose a voltage domain and change the voltage for all the cores in that domain.

The SCC has 24 frequency domains, one for each core. You can change the frequency of each individual core. However, the power management calls in the RCCE API do not currently support the control of all frequency domains independently. Instead, RCCE only allows stepping of the frequency on the cores within a voltage domain. Effectively for RCCE, frequency and voltage domains coincide and are jointly called *power domains*.

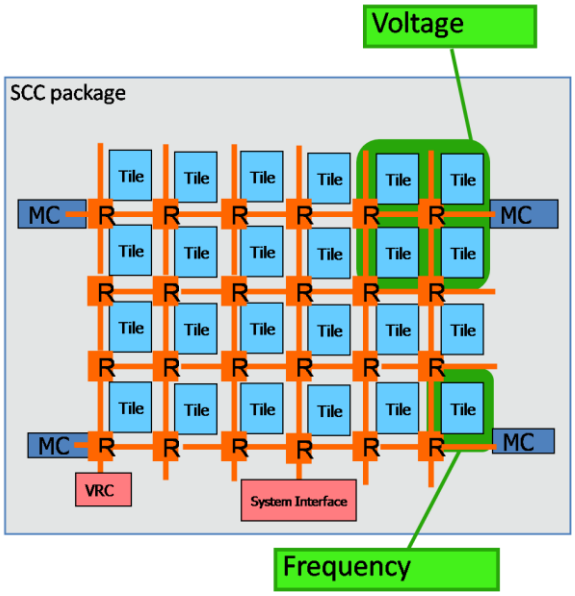

Figure 12: Voltage and Frequency Domains

One usage possibility is initially to set up areas of voltage and frequency values and then move tasks to the area where it has the best performance while minimizing power consumption. Because tasks share no state, they can easily be mapped to different cores.

You can control the voltage of a voltage domain by writing the VRC control register. You can change the frequency by writing the GCU (Global Clock Unit) register. The best way, however, is to use the power management API that RCCE provides. See Section 8.7 Power Management for information about using the RCCE power management functions.

6 Some Simple SCC Programs

This section shows how to run some simple C programs on the SCC platform. These programs are not RCCE programs, so you should not load them with the **rccerun** script that comes with RCCE. **rccerun** makes some assumptions that do not hold for arbitrary SCC programs. Rather you should use **pssh** (the parallel ssh). You can use Ubuntu's Synaptic Package Manager to install pssh. Note that the pssh executable is then called **parallel-ssh**, and you must create a link called **pssh**. RCCE expects the executable to be called **pssh**.

This section shows two examples. The first is just a "hello world." Its purpose is to show how you would create the **pssh** hosts file and then compile and load the application. The second example is also very simple, but more realistic. It shows how you would read and write a core configuration register.

6.1 Hello World

This example assumes that you've loaded SCC Linux on at least two cores, namely cores 00 and 01.

Make a **pssh** hosts file. Name it anything you want. The format must be as follows

```
rck00 root
rck01 root
rck02 root
rck03 root
rck04 root
rck05 root
    :
```

Each line has two fields. The first identifies the core as **rck***processorID*, and the second identifies the user on the cores. The user is always root.

The code is in the file **SCChello.c**. It just includes **stdio.h** and has a **main()** that calls **printf()**. Compile it as follows.

```
icc -DSCC -static -mcpu=pentium -gcc-version=340 helloSCC.c
```

The macro **SCC** specifies that you are compiling for SCC hardware. The switch **-mcpu=pentium** indicates that you are compiling for the P54C architecture. For this reason, you must also use an older version of **icc** (8.1.038).

```
YourUsername@YourComputer:/shared/YourUsername> which icc
/shared/icc-8.1.038/bin/icc
YourUsername@YourComputer:/shared/YourUsername>
```

icc/icpc does require that you have **gcc/g++** installed. The version of **icc/icpc** that you need for SCC is validated for gcc 3.4.0. You can try a later version of **gcc**. The MCPC that this example ran on has **gcc** 4.1.2; but the **icc/icpc** 8.1 validation stopped at **gcc** 3.4.0. You must use the switch **-gcc-version=340** to ensure that **icc/icpc** does not use any **gcc** features beyond 3.4.0.

Copy the resulting **a.out** to the **/shared** directory on the MCPC. Typically, users create a subdirectory under **/shared** and name it with their username. Then, use **pssh** to load **a.out**

on SCC cores.

You must put your executable under **/shared** for it to be recognized by the cores. The directory /shared is mounted on the cores. In addition, it's best to let the RCCE configure script create your directory under **/shared** for you. It puts some scripts and files in there that you will find useful later.

The RCCE configure script is part of the RCCE download. The RCCE download is stored in the trunk of our public SVN at http://marcbug.scc-dc.com/svn/repository/trunk/rcce/ . To download RCCE and run configure, issue the following commands. The directory name **sandbox** is arbitrary.

```
cd
mkdir sandbox
svn co http://marcbug.scc-dc.com/svn/repository/trunk/rcce/
cd rcce
./configure SCC_LINUX
```

How many cores you load the program on depends on the pssh hosts file and the **pssh -p** switch. In this example the pssh hosts file (called **pssh.hosts**) has only two lines

```
rck00 root
rck01 root
```

The **-p** switch specifies the number of threads. The **pssh** command runs one thread per core. So with two lines in the pssh hosts file, you see "hello" printed by each core.

If the pssh hosts file had four entries, you would see "hello" printed four times, but the programs would run first on two cores and, when the first two cores complete, the program then runs on the next two cores in the list. This is different from **-p 4** because then the program would run on four cores at once.

The switch **-p** actually specifies the maximum number of threads. So, if your pssh hosts file has two lines and you enter **-p 4**, the program would still load and run, but run on only two cores.

```
YourUsername@YourComputer:/shared/YourUsername> pssh -h pssh.hosts -t
-1 -P -p 2 /shared/YourUsername/a.out
rck00: hello
rck00: [1] 08:43:43 [SUCCESS] rck00 22
rck01: hello
rck01: [2] 08:43:43 [SUCCESS] rck01 22
YourUsername@YourComputer:/shared/YourUsername>
```

Note that the **pssh** command line specifies the full pathname for **a.out**, even though **a.out** is in the working directory. If you don't specify the full pathname, you get an error.

```
YourUsername@YourComputer:/shared/YourUsername> pssh -h pssh.hosts -t
-1 -P -p 2 a.out
rck00: sh: rck01: sh: rck00: rck00: a.out: not found
rck01: a.out: not found
rck01: [1] 08:46:51 [FAILURE] rck00 22 Received error code of 127
[2] 08:46:51 [FAILURE] rck01 22 Received error code of 127
YourUsername@YourComputer:/shared/YourUsername>
```

The **-t** switch specifies the timeout in seconds, and -1 means it never times out. The **-P** switch specifies that the program prints output as it is received.

6.2 Reading and Writing Core Configuration Registers

This example shows how a core program can access core configuration registers. It can access its own configuration registers as well as those of other cores using memory-mapped I/O. The core program performs memory-mapped I/O in the standard Linux way using the **mmap()** function. A good reference for how to perform memory-mapped I/O is *Advanced Programming in the Unix Environment* by W. Richard Stevens and Stephen A. Rago.

The configuration registers for each tile are given a base address in the core's LUT. The RCCE header file **config.h** defines macros for these base addresses. Realize, however, that these are the base addresses that result from the default LUT configuration. Note that there is a special address that a core can use to identify its own base address.

The base address for the configuration registers for the tile at (x=0, y=0) is 0xe0000000. The configuration registers for each tile are offset by 0x01000000 from 0xe0000000 as you travel along the x axis. Following this convention, the base address for the tile at (x=1, y=0) is 0xe1000000, that for the tile at (x=2, y=0) is 0xe2000000, etc. The tile after (x=5, y=0) is (x=0, y=1), etc. Continuing with this method, the base address for the final tile at (x=5, y=3) is 0xf7000000. Table 5.shows the base addresses for the configuration registers for the SCC tiles.

Base Address	Tile (x,y)
F8000000	Base address for Calling Core
F7000000	System Configuration Registers -- Tile (x=5,y=3)
F6000000	System Configuration Registers -- Tile (x=4,y=3)
:	:
E2000000	System Configuration Register s-- Tile (x=2,y=0)
E1000000	System Configuration Registers -- Tile (x=1,y=0)
E0000000	System Configuration Register s-- Tile (x=0,y=0)

Table 5 Base Addresses for Core Configuration Registers

The base address 0xf8000000 is the special one. When a core specifies this base address, it specifies its own base address. A core can reference its own base address in this way to read its own TileID register and obtain its own (x,y) coordinates, tileID, coreID, and processorID. Appendix contains a sample code listing that a core can use to read its TileID register. The

The key points of that program are shown below. For the file descriptor of the file to be mapped, use the device **/dev/rckncm**. Open the device **/dev/rckncm** and get its file descriptor.. Call **mmap()** and specify this file descriptor and map a page. Dereference the returned pointer to get the value of the TileID register.

```
typedef volatile unsigned char* t_vcharp;
int          tileID;
unsigned int alignedAddr, pageOffset;
t_vcharp     MappedAddr;
             :
if ((NCMDeviceFD=open("/dev/rckncm", O_RDWR|O_SYNC))<0) {
   perror("open"); exit(-1);
}
```

```
alignedAddr = 0xf8000000;
pageOffset  = 0x100;

MappedAddr = (t_vcharp) mmap(NULL, PAGE_SIZE, PROT_WRITE|PROT_READ,
   MAP_SHARED, NCMDeviceFD, alignedAddr);
if (MappedAddr == MAP_FAILED) {
   perror("mmap"); exit(-1);
}
tileID = *(int*)(MappedAddr+pageOffset);
            :
```

Compile with **icc**. Then copy the resulting executable to **/share/YourUsername**.

```
icc -DSCC -static -mcpu=pentium -gcc-version=340 readTileID.c
```

As with the "hello-world" example, the file **pssh.hosts** determines what processors the program runs on. For this example, **pssh.hosts** looks as follows.

```
rck10 root
rck11 root
```

This file causes the program to run on processors 10 and 11. Run the program with the **pssh** command.

```
YourUsername@YourComputer:/shared/YourUsername> pssh -h pssh.hosts -t
-1 -P -p 2 a.out
```

The output called tileID prints first in hexadecimal and then in decimal.

```
rck11: tileID = 29 41
rck11: [1] 17:33:16 [SUCCESS] rck11 22
rck10: tileID = 28 40
rck10: [2] 17:33:16 [SUCCESS] rck10 22
```

From the tileID that is returned you can obtain the tile's (x,y) coordinates and the core's coreID by shifting and masking. Then, the core's tileID is 16*y + x. Its processorID is (x+(6*y))*2 + coreID.

The (x,y) coordinates for this example are (5,0). The TileID register has y in bits 10:07, x in bits 06:03 and the coreID (which is 0 or 1) in bits 02:00. To be completely accurate, bits 02:00 contain the subID of the requesting agent; and if the requesting agent is a core, its subID is the coreID. Note, though, that if a core reads the TileID register of another core, it is the core doing the read that is the requesting agent.

If you are using the sccGui to access a TileID register, the requesting agent is the System Interface whose subID is 101b.

The TileID register for processorID 10 is 0x28, and the Tile ID register for processorID 11 is 0x29. Table 6 shows the bit pattern of the TileID register and how the values 0x28 and 0x29 populate its fields

Y	X	CoreID	
1 0987	6543	210	Hex Value
0000	0101	000	0x28
0000	0101	001	0x29

Table 6 Bit Pattern for the TileID Register

7 Building RCCE

Get the latest RCCE source code by checking RCCE out from an Intel-provided SVN repository. You may choose to **svn export** rather than **svn co** if you don't want all those .svn directories.

Type **./configure** <*PLATFORM*> where <*PLATFORM*> is either **SCC_LINUX** or **emulator**. Use **SCC_LINUX** when you are building for SCC hardware; use **emulator** when you are building the RCCE emulator.

The command is silent. The option **SCC_LINUX** creates your username directory under **/shared** and populates it with the file **allhosts** and the two scripts: **killcorePIDS** and **killit**. Use the **killit** script to kill programs running on the cores. **Control-C** issued from the MCPC only kills the **pssh** script, not a core program.

Do not create **/shared** on your own. By using the **configure** script, you get all the needed files under **/shared**.

The **configure** script also modifies the **common/symbols** file to make it appropriate for your installation. The **symbols** file sets compiler flags and determines whether to build RCCE for SCC hardware or the emulator. The default is to build all the libraries for the emulator.

Do not edit the **symbols** file. It is constructed by **configure** from **symbols.in**. If you want to have a new **symbols** file, edit **symbols.in** and reconfigure.

If you build the RCCE emulator on the MCPC, be sure to use the native compiler and not the cross compiler.

7.1 Building the RCCE Emulator

First, **edit common/symbols.in** to choose the compiler you want. Don't use the cross compiler that comes with an SCC system. The cross compiler is intended to build applications that run on SCC hardware. The emulator runs on a modern Linux system, typcially the MCPC, It should run on Windows and MAC OS as well, but those platforms don't get a lot of use.

Then, run the **configure** script as

```
./configure emulator
```

If you want to include the capability for power management, run the **configure** script as

```
./configure emulator ADD_POWER_API
```

One of the results of the **configure** script is to create a **common/symbols** from common/symbols.in. It's not good practice to edit **common/symbols** directly because the next time you run **configure**, your edits will be overwritten.

To build all the libraries for the emulator, use the **makeall** command . This means that if you are on your Linux desktop and you just type **./makeall**, you will build the following libraries under **bin/OMP**.

```
$ ls -l
total 268
-rw-r--r-- 1 user group 65538 2011-09-06 14:50 libRCCE_bigflags_gory_nopwrmgmt.a
-rw-r--r-- 1 user group 64768 2011-09-06 14:50 libRCCE_bigflags_nongory_nopwrmgmt.a
-rw-r--r-- 1 user group 66546 2011-09-06 14:50 libRCCE_smallflags_gory_nopwrmgmt.a
-rw-r--r-- 1 user group 65760 2011-09-06 14:50 libRCCE_smallflags_nongory_nopwrmgmt.a
```

Note that there are four libraries. The power management libraries in this example are not included. You have a library with bigflags and a library with smallflags. Each of those can have either a gory or a nongory interface.

Note that RCCE routines may use some C99 syntax, and so you may need to modify the **CCOMPILE** line in **common/symbols.in** to include the switch **-std=c99** so that these routines can build. This is the **CCOMPILE** definition when **OMP_EMULATOR** is 1, which is its default.

Refer to Section 7.3 RCCE Build Options for information about the difference between bigflags and smallflags and the difference between the gory and nongory interfaces.

If you issue just **make**, you get the library **libRCCE_bigflags_nongory_nopwrmgmt.a** in **bin/OMP**.

7.2 Building RCCE for SCC Hardware

Build the RCCE library for SCC hardware on the MCPC. You don't have to build it on the MCPC, but if you do, you are assured of access to the correct cross-compiler. You are building for the SCC cores, not the MCPC.

Check out (or export) RCCE source from the SCC public svn repository. If you issue configure with no options, you get the list of options as follows.

```
$ ./configure
Usage: ./configure emulator
       ./configure SCC_LINUX
       ./configure SCC_BAREMETAL
See README for power management options
```

Choose **SCC_LINUX**. This option will create the RCCE libraries for SCC hardware without the power management API. The libraries are created in the directory **bin/SCC_LINUX**. Ensure that the crosscompilers (both **icc** and **gcc**) is in your path and that **LD_LIBRARY_PATH** is defined.

```
$ which icc
/opt/icc-8.1.038/bin/icc
$ which gcc
/opt/i386-unknown-linux-gnu/bin/gcc
```

```
$ env |grep LD_
LD_LIBRARY_PATH=/opt/icc-8.1.038/lib:/opt/i386-unknown-linux-gnu/lib
```

RCCE configured for baremetal has not been tested for the latest RCCE software. The baremetal framework most users now use is the one contributed by ETI (http://www.etinternational.com/). Refer to http://communities.intel.com/message/108074#108074 .

The power mnagement API remains an experimental feature. Please consult the RCCE **README** file. To build RCCE libraries with the power management API, configure as

```
./configure SCC_LINUX ADD_POWER_API
```

Refer to Section 7.3 RCCE Build Options for information about the difference between bigflags and singlebitflags, the difference between the gory and nongory interfaces, and whether to include the power management API.

7.3 RCCE Build Options

The gory and nongory interfaces are aptly named. If you want to write RCCE applications and get down into the nitty gritty of how message passing is implemented, use the gory interface (**API=gory** on the **make** line). Otherwise choose, the nongory interface (**API=nongory** on the make line, which is the default).

Flags are used to coordinate interaction between units of execution. You can choose flags to have low latency and be somewhat wasteful of memory or flags that have a higher latency and consume less memory. The memory referred to here is message passing buffer memory.

The first choice (lower latency, higher memory use) occurs when you specify **SINGLEBITFLAGS=0** on the **make** command line. With bigflags, each flag takes up a byte; there are eight flags per 32-byte cache line. The second choice (higher latency, lower memory use) occurs when you specify **SINGLEBITFLAGS=1** on the **make** command line. With singlebitflags, flags are stored as a single bit.

Having the power management routines in the library does not affect the performance of the RCCE library, but sometimes you want the library to include only what you plan on using.

8 Running RCCE Applications

The SCC Platform Overview showed how to run an RCCE example using the RCCE emulator. The example was pingpong, which just sends messages back and forth between two cores.

The RCCE distribution contains several examples in its **apps** directory. The point of entry for all units of execution (UEs) into a RCCE applications is **RCCE_APP()**. Recall that a UE refers to a process, thread or other agent of execution that moves the program counter forward. For RCCE applications, the number of UEs is the number of cores (when running on SCC hardware) or the number of simulated cores (when running under the RCCE emulator).

The call **RCCE_APP()** is really intended for the RCCE emulator, which runs on top of

OpenMP. When you are programming for actual SCC hardware, you can replace **RCCE_APP()** with **main()**. You do not have to perform this replacement because the file **RCCE.h** does it for you as shown below. When you are running the RCCE emulator, **_OPENMP** is defined.

```
// little trick to allow the application to be called "RCCE_APP" under
// OpenMP, and "main" otherwise
#ifndef _OPENMP
  #define RCCE_APP main
#endif
```

To start a RCCE application, use **rccerun**. The options for **rccerun** are as follows.

```
rccerun [-emulator] -nue nbrUEs [-f hostfile] [-clock GHz] executable
[parameters]
```

You must specify the number of UEs and a hostfile. The hostfile is a file that lists the processorIDs of the cores that **rccerun** will load the application on. It contains one line for each SCC core as follows.

```
00
01
02
03
:
```

When **rccerun** loads your RCCE application, it chooses the number of cores specified by **-nue** starting with the processorID on the first line.

-clock specifies the frequency that RCCE uses for timing measurements. It does not change the actual frequency of the SCC tiles. If you specify a frequency with **-clock** on the **rccerun** command line that is different from the actual tile frequency, the timing results returned by RCCE examples will be incorrect.

rccerun either uses the RCCE emulator to run your application or loads your application on SCC hardware, depending on what you set **<PLATFORM>** to when you ran **configure**.

Specify the name of your RCCE executable. If this RCCE executable is not in your current working directory, give its pathname, which can be relative.

If you want to execute your RCCE program on SCC hardware, the SCC must have access to the executable. Users usually place the RCCE executable under **/shared** on the MCPC. The directory **/shared** exists on the MCPC and is mounted on the SCC cores.

If you've configured for the emulator and neglect to specify **-emulator** on the **rccerun** command line, you get Error Code 127.

Be sure to use the **rccerun** that is created in the same directory as the **configure** script. Put this **rccerun** in your path. If you use a **rccerun** from a different RCCE build, you may see an Error Code 139 when you run your RCCE application.

8.1 Characteristics of RCCE Programs

From a programmer's point of view, RCCE applications are message passing applications.

Because of the lack of cache coherence among the cores, the programming model is most naturally and most efficiently based on the ability to send messages between the cores.

Recall that SCC has an on-chip message-passing buffer (MPB). Each of the 24 tiles has 16KB of MPB, totaling 384KB for the chip itself. Each of the 48 cores is assigned 8KB of this MPB, but each core has access to the entire MPB. A core can send a message to another core by moving data from its own L1 cache to the MPB. Then, the receiving core can take the data from the MPB and move it into its own L1 cache.

Note that messages are passed from one core to another without a core having to use any off-chip memory. Note also that messages bypass a core's L2 cache. The sending core *puts* the message from its L1 cache into the sending core's MPB. Then, the receiving core *gets* the message from the sending core's MPB. This is the "put" model.

The "push" model is when the sending core puts the message from its L1 cache into its own MPB. Then the receiver gets the message from the sener's MPB.

However, referring to the MPB as sender's MPB and receiver's MPB is somewhat artificial because the MPB address space is accessible by all cores. RCCE manages the MPB by assigning 8KB regions to each core, but any core can write anywhere in the MPB.

As described above, RCCE is based on one-sided put and get primitives. RCCE also provides two-sided synchronous message passing calls. Internally, these high-level message passing calls are implemented as one-sided primitives with flags to control access to the MPB.

8.2 Two Important Cautions When Using RCCE

The first cautions against assuming that the initial state of the message passing buffer and test-and-set registers are clean. The second cautions against assuming that the synchronizing calls with empty messages actually synchronize.

8.2.1 Initial State of the Message Passing Buffers and Test-and-Set Registers

There is no guarantee that the MPBs are in a clean state, at the beginning of a RCCE execution. You can explicitly wipe the MPBs by executing **`mpb -c`** on the cores. Run it on each core whose MPB you want to clear. You can also use the SCC GUI and select Tools➔Clear MPB(s).

Similarly, there is no guarantee that the test-and-set registers are in a clean state. You can reset the test-and-set registers by executing **`mpb -cl`** on the cores. Run it on each core whose test-and-set register you want to clear.

If the application dies and needs to be killed, the state of all registers and on-chip memory is indeterminate. However, even in a correctly executing code, it is possible to leave debris in the MPBs and the test-and-set registers.

8.2.2 Empty Messages do not Synchronize

Note that if the synchronizing calls, **`RCCE_send()`** and **`RCCE_recv(),`** have empty messages, they are not synchronizing. If the message size is zero, the send and receive calls are

effectively no-ops and hence senders and receivers do not need to be matched.

This is different from MPI where even an empty message is not really empty. In MPI, the payload is accompanied by a header, allowing MPI programmers to use an empty message for synchronization. RCCE communication calls only have payload. When the message is null, there is no need to communicate and no synchronization occurs. With RCCE, do not use **`RCCE_send()`** and **`RCCE_recv()`** with an empty message for synchronization

8.3 RCCE has Basic and Gory Interfaces and Power Management

The RCCE libraries are distributed as C source code. Recall from Building RCCE that you can build the RCCE library with either the basic or the gory interface. The gory interface exposes the RCCE one-sided primitives. The gory interface is not a strict superset of the basic interface. It contains some routines with the same name as those in the basic interface but with a different set of parameters. These routines are called auxiliary routines, and they are made up from some of the elementary gory routines.

Table 7 lists the RCCE calls belonging to the basic and gory interfaces. This table does not contain the entire set of RCCE calls. RCCE also contains some power management routines, which are shown in Table 8 RCCE Power Management Routines

The remainder of this section describes the RCCE routines in more detail.

Basic	Gory
Core	**Core**
RCCE_init()	RCCE_int()
RCCE_finalize()	RCCE_finalize()
RCCE_ue()	RCCE_ue()
RCCE_debug_set()	RCCE_debug_set()
RCCE_debug_unset()	RCCE_debug_unset()
RCCE_error_string()	RCCE_error_string()
RCCE_wtime()	
RCCE_comm_rank()	
RCCE_comm_size()	
Communication	**Communication**
RCCE_send()	Auxiliary RCCE_send()
RCCE_recv()	Auxiliary RCCE_recv()
RCCE_recv_test()	Auxiliary RCCE_recv_test()
RCCE_reduce()	RCCE_reduce()
RCCE_allreduce()	RCCE_allreduce()
RCCE_bcast()	RCCE_bcast()
RCCE_comm_rank()	RCCE_comm_rank()
RCCE_comm_size()	RCCE_comm_size()
RCCE_comm_split()	RCCE_comm_split()
	RCCE_put()
	RCCE_get()
	RCCE_flag_write()
	RCCE_flag_read()
Synchronization	**Synchronization**
RCCE_barrier()	RCCE_barrier()
RCCE_fence()	RCCE_fence()
	RCCE_wait_until()
Memory Management	**Memory Management**
RCCE_shmalloc()	RCCE_shmalloc()
RCCE_shfree()	RCCE_shfree()
RCCE_shflush()	RCCE_shflush()
RCCE_malloc()	RCCE_malloc()
	Auxiliary RCCE_malloc_request()
	RCCE_free()
	RCCE_flag_alloc()
	RCCE_flag_free()

Table 7 RCCE Calls Belonging to the Basic and Gory Interfaces

RCCE Power Management
RCCE_power_domain()
RCCE_power_domain_master()
RCCE_power_domain_size()
RCCE_iset_power()
RCCE_wait_power()
RCCE_set_frequency()

Table 8 RCCE Power Management Routines

8.4 The STENCIL Example

The next two sections illustrate the use of the RCCE basic and gory interfaces. The RCCE release comes with a number of examples in the **`apps`** directory. The STENCIL example is in **`apps/STENCIL`**.

<TBD>
Provide an overview of what the STENCIL example actually does … what problem is it intended to solve?
</TBD>

RCCE provides two versions of the STENCIL example. One (**`RCCE_stencil_synch.c`**) uses the basic interface, and the other (**`RCCE_stencil.c`**) uses the gory interface.

In STENCIL, a matrix is partitioned among the cores. The first and last rows of the array are fixed. The first row is fixed at 1.0, and the last row is fixed at 2.0. That's how STENCIL prints out the matrix, but internally, a core refers to other cores at its right and left. So you can also think of the matrix as having its first column fixed at 1.0, and its last column fixed at 2.0.

`RCCE_num_ues()` returns the number of cores (n) participating in the calculation. A core's ID (also called its sequence number or its rank) goes from 0 to **`RCCE_num_ues()`** -1 (n-1).

The matrix never exists completely on one core. In STENCIL, the matrix **`a`** is declared as a one-dimensional array of dimension **`NX*NY`**. There are **`NX`** columns on each core. For each core, the row starts at **`a[offset]`**. For core 0, **`offset`** is 0; for core n-1, **`offset`** is **`NX*(NY-1)`**.

Consider the example when **`NX`**=8, **`NY`**=10, and the number of cores is 4. This array is distributed among the four cores as follows.

```
Core 0     9 rows (elements of top row are fixed 1.0)
Core 1     8 rows
Core 2     8 rows
Core 3     9 rows (elements of bottom row are fixed 2.0)
```

All other elements are zero. As the calculation proceeds and the stencil is applied, the 1.0 and 2.0 flow toward the center and approach stabilization.

8.5 RCCE Basic

To build the basic version of STENCIL, enter the directory **apps/STENCIL** and type

```
make stencil_synch
```

To run the resulting executable on four cores and perform 50 iterations, enter

```
rccerun -nue 4 -f ../../hosts/rc.hosts stencil_synch 50
```

The default number of iterations is 10, but you can override this default from the command line as shown above.

The command **rccerun** also has an option that specifies the clock. The default is 1GHz. If the cores were running at 748 MHz, you would add **-clock 0.748** to the **rccerun** invocation. The stencil programs print out some timing information and use the value provided with **-clock**.

<TBD>
Provide a description of use of RCCE_send() and RCCE_recv() and describe how/why they are synchronous.
</TBD>

8.6 RCCE Gory

To build the gory version of STENCIL, enter the directory **apps/STENCIL** and type

```
make API=gory stencil
```

To run the resulting executable on four cores and perform 50 iterations, enter

```
rccerun -nue 4 -f ../../hosts/rc.hosts stencil 50
```

<TBD>
Provide a description of use of RCCE_put(), RCCE_get(), RCCE_flag_write(), RCCE_wait_until(), RCCE_flag_alloc().
</TBD>

8.7 Power Management

Although you can manage SCC power by writing the VRC configuration register directly through memory-mapped I/O, it's highly recommended that you use the RCCE power management calls. When you use these RCCE calls, you run less risk of crashing the SCC platform. RCCE provides you with power management capabilities, but it does not provide you with access to everything that the VRC can do. Note that the VRC (voltage regulator controller) is called the RPC (Rock Creek power controller) in some older documentation.

For information about how to access the VRC directly (not through RCCE), refer to How to Change the Voltage Directly.

To use RCCE's power management functions, you must first configure RCCE for power management and then build RCCE

```
./configure SCC_LINUX ADD_POWER_API
```

```
./makeall
```

Then, when you compile a RCCE application that uses the power management API, specify **PWRMGMT=1** on the make line as in

```
make PWRMGMT=1 powertest
```

RCCE assumes that the 2x2 voltage domains and the frequency domains are identical. RCCE does not recognize the "all-mesh" power domain. As far as RCCE is concerned, there are six 2x2 power domains, labeled 0 to 5.

RCCE assigns a single core in a power domain as a power domain master. This designation is not user configurable. Only the power domain master can communicate with the SCC power management facility. Power management calls issued by other cores are ignored and return immediately with the return value **RCCE_SUCCESS.**

For maximum impact on the power budget, RCCE modifies the frequency in concert with the voltage, through the combined power command, **RCCE_iset_power()**. The input to the function specifies the desired tile frequency divider (an integer ranging from 2 to 16).

The frequency is defined relative to a global reference clock that is set when the SCC platform first starts up. This reference clock can vary, but in almost all cases it is 1.6Ghz. Changing it to a different value is an advanced procedure that is not recommended.

RCCE_iset_power() sets the tile frequency to the reference clock divided by the supplied divisor. For example, if the divider is 4, the tile frequency is then 1.6GHz/4 = 400MHz. **RCCE_iset_power()** then determines the lowest voltage level that is consistent with the input value of the frequency divider and initiates the voltage change. **RCCE_iset_power()** provides output values that define the actual settings for the frequency divider and the voltage level.

The returned voltage level is an integer from 0 to 6. The actual voltage (in volts) is 0.7 + voltage_level * 0.1. This means that the minimum voltage is 0.7v and the maximum voltage is 1.3v. These are nominal voltages.

Because changing the voltage has such a high latency, **RCCE_iset_power()** is not a blocking call. When you issue **RCCE_istep_power()**, you get a request ID. The call does not block, and your program continues. Later you can issue the call **RCCE_wait_power()** and specify a request ID.

The intent is for **RCCE_wait_power()** to block until the preceding request from **RCCE_iset_power()** is satisfied. ." What the call initiallydid was just reissue the power change request; the belief was that it would block until the first request was processed. Later some work showed that this was not sufficient ... that one must issue the request yet again and wait for the third request to return. Look inside **RCCE_power_management.c** for details.

RCCE_set_frequency_divider() sets the frequency of the cores in the domain of the calling core without changing the voltage. As with **RCCE_iset_power()**, you supply a frequency divider. **RCCE_set_frequency_divider()** will not let you set the frequency to a value that is too high for the current voltage.

Because changing the frequency has a low latency, **RCCE_set_frequency()** is a blocking

call. As with **RCCE_iset_power()**, the call only has an effect when executed by the power domain master. Non-master cores do nothing and just return with **RCCE_SUCCESS**.

The file Using RCCE Power Management contains an example program. Note that RCCE and the SCC label the power domains differently as shown in Figure 13.

RCCE: 3 SCC: 0	RCCE: 4 SCC: 1	RCCE: 5 SCC: 3
RCCE: 0 SCC: 4	RCCE: 10 SCC: 5	RCCE: 2 SCC: 7

Figure 13: RCCE vs SCC Power Domains

The consequence is that if you use the BMC **status** command to read the voltage, VCC4 refers to RCCE power domain 0. For example, if you use RCCE to change the voltage in RCCE power domain 0 to 0.8 volts, the sccBmc –c status command returns

```
Tertiary supplies:
  OPVR VCC0: 1.0948 V
  OPVR VCC1: 1.0939 V
  OPVR VCC2: 1.0913 V
  OPVR VCC3: 1.0936 V
  OPVR VCC4: 0.8421 V
  OPVR VCC5: 1.0897 V
  OPVR VCC7: 1.0870 V
```

8.7.1 Power Domains

RCCE provides three informational calls that deal with power domains.

- **RCCE_power_domain()** returns the number of the power domain that contains the calling core. The power domain number does not change; it is hard linked to the position of the core in the mesh. As shown in Figure 12 and Figure 13, there are six power domains, and each contains four tiles in a 2x2 array. The power domain in the lower left is 0. The domain immediately to its right is 1 and then 2. The upper row contains domains 3, 4, and 5, with 5 being the domain in the upper right.
- **RCCE_power_domain_master()** returns the rank of the domain master in the local power domain. RCCE refers to a core's processorID as its sequence number or its rank. Table 2 shows how the processorIDs are arranged in the mesh. These processorIDs start with 0 and end with 47.
- **RCCE_power_domain_size()** returns the number of cores in the local power domain that are participating in the computation. Note, however, that when you change the power for cores in a domain, you change the power for all the cores, even those that are not participating in the computation.

8.7.2 Changing the Power

RCCE_iset_power() takes four parameters. The first is an input parameter called **Fdiv** that specifies the desired frequency divider. The second is an output parameter that will contain a request ID. The third and fourth parameters are also output parameters that tell you what

the frequency divider and voltage level got set to. Most likely this output frequency divider is equal to your input frequency divider.

Table 9 shows the maximum frequency allowed for a particular voltage level. For example, if the voltage is 1.1v, the voltage level is 4, and a frequency above 875MHz may damage the chip. If the voltage is 0.9v, the voltage level is 2, and a frequency above 644MHz may damage the chip.

Voltage Level	Voltage (volts)	Maximum Frequency (MHz)
0	0.7	460
1	0.8	598
2	0.9	644
3	1.0	748
4	1.1	875
5	1.2	1024
6	1.3	1198

Table 9 Voltage and Frequency Values

When you start up the SCC platform with the sccGui, you can choose to have the tiles run at either 800MHz or 533MHz. These two values correspond to frequency dividers of 2 and 3 respectively. Table 10 lists the tile frequencies corresponding to all the possible frequency dividers.

With `RCCE_iset_power()`, you specify a desired frequency divider. The call creates a temporary variable equal to the reference clock divided by that frequency divider. The call then traverses Table 9 starting with voltage level 0 and finds the first voltage level whose maximum frequency is greater that the value of this temporary variable. The call then sets the voltage to that value and sets its output values to that voltage level and frequency divider.

For example, if you choose a frequency divider of 3, the temporary variable equals 533MHz. The call sets the voltage level to 1. If you choose a frequency divider of 2, the temporary variable equals 800MHz. The call then sets the voltage level to 4.

A consequence of the way RCCE sets voltage and frequency is that if you set the frequency divider to the very same value you initialized with, the voltage decreases, and you cannot return the voltage to its original value with RCCE. You can return to the default voltage by cycling the SCC power or setting the voltage directly.

Tile Frequency (MHz)	RCCE Frequency Divider	RCCE Voltage Level
800	2	4
533	3	1
400	4	0
320	5	0
266	6	0
228	7	0
200	8	0
178	9	0
160	10	0
145	11	0
133	12	0
123	13	0
114	14	0
106	15	0
100	16	0

Table 10 Tile Frequencies and RCCE Frequency Dividers

Here is another example. Assume that you issue

```
RCCE_iset_power(4, &request, &newVoltageDiv, &newFreqDiv)
```

Then, the temporary variable equals 400MHz, **`newVoltageDiv`** is 0, and **`newFreqDiv`** is 4. The actual voltage is 0.7v + 0*0.1v = 0.7v. The actual frequency is 1.6GHz/4 = 400MHz.

When you go to a power level that has a larger frequency, RCCE increases the voltage first and then the frequency. When you go to a power level that has a lower frequency, RCCE decreases the frequency first and then the voltage.

Only one **`RCCE_istep_power()`** can be "in flight" at the same time for a power domain. If you issue another **`RCCE_istep_power()`** before the first one is satisfied, the second request is denied, and it returns an error code.

8.7.3 Changing the Frequency

Use **`RCCE_set_frequency_divider()`** to change the frequency. Because changing the frequency has low latency, the call is blocking. The call takes two parameters. The first is an input parameter that is the requested frequency divider; the second is an output parameter that is the frequency divider that actually results. Table 10 lists the available frequency dividers and the tile frequencies they correspond to.

A frequency divider less than 2 returns an error. A frequency divider greater than 16 gets set to 16.

9 Building your own Linux Image

Intel Labs provides a Linux image of SCC Linux that you can load on the cores. This default Linux image is called **`linux.obj`** and is stored in **`/opt/sccKit/current/resources`** on the management console (MCPC). It is a hex-encoded ASCII file describing the operating environment for SCC.

You have the ability to create your own Linux image. You may want essentially the default Linux image, but you want to add other utilities or apply a kernel patch.

For information about how to build a custom SCC Linux refer to How to Build SCC Linux 1.4.1.x .

10 Appendix

10.1 SCChello.c

```
#include <stdio.h>
main() {
   printf("hello\n");
}
```

10.2 readTileID.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

#define CRB_OWN   0xf8000000
#define MYTILEID  0x100

main() {

typedef volatile unsigned char* t_vcharp;

int PAGE_SIZE, NCMDeviceFD;
// NCMDeviceFD is the file descriptor for non-cacheable memory (e.g. config regs).

unsigned int result, tileID, coreID, x_val, y_val,
   coreID_mask=0x00000007, x_mask=0x00000078, y_mask=0x00000780;

t_vcharp     MappedAddr;
unsigned int alignedAddr, pageOffset, ConfigAddr;

   ConfigAddr = CRB_OWN+MYTILEID;
   PAGE_SIZE  = getpagesize();

   if ((NCMDeviceFD=open("/dev/rckncm", O_RDWR|O_SYNC))<0) {
    perror("open"); exit(-1);
   }

    alignedAddr = ConfigAddr & (~(PAGE_SIZE-1));
    pageOffset  = ConfigAddr - alignedAddr;

    MappedAddr = (t_vcharp) mmap(NULL, PAGE_SIZE, PROT_WRITE|PROT_READ,
       MAP_SHARED, NCMDeviceFD, alignedAddr);

   if (MappedAddr == MAP_FAILED) {
      perror("mmap");exit(-1);
   }

  result = *(unsigned int*)(MappedAddr+pageOffset);
  munmap((void*)MappedAddr, PAGE_SIZE);

  printf("result = %x %d \n",result, result);

   coreID = result & coreID_mask;
   x_val  = (result & x_mask) >> 3;
   y_val  = (result & y_mask) >> 7;
   tileID = y_val*16 + x_val;

   printf("My (x,y) = (%d,%d)\n", x_val, y_val);
```

```
    printf("My tileID = 0x%2x\n",tileID);
    printf("My coreID = %1d\n",coreID);
    printf("My processorID = %2d\n",(x_val +(6*y_val))*2 + coreID);

}
```